ECE 385 Final Project Report

Heyi Tao (heyitao2)
Hongshuo Zhang (hz13)
Section ABD
05/05/2020
TA: Nicholas Cebry, Wenjie Pan

Introduction

We implemented a simpler version of a popular arcade game called Battle City. Our game supports two game modes -- one player mode and two player mode. In one player mode, the player can choose up to 2 enemies to play against. In two player mode, two players are expected to play against each other. In both game modes, each tank starts with five remaining lives and the player wins if all other tanks have zero lives and the player loses if all five of their available lives are lost.

Written Descriptions of Final Project

Our game can be roughly broken up into four major components:

- Start Screen
    - This component is the first component of the game. It has a cool background image and it prompts the player(s) to choose the game mode they are interested in.
    - The default game mode is set to one player mode. However, the player can change the game mode by one of the following keys. (W, S, up, and down).
    - A cute tank points to the game mode that the player has selected.
    - The game is stuck in this component until one of the following keys (Enter or space) is pressed.

Figure 1: The start screen of the our game
Currently, the first player mode is selected

- Enemy Selector
    - This component is the second component of our game. **This component can be only reached if the one player mode is chosen**. This component also has a very cool background image that reads ENEMY and it prompts the user to choose the number of enemies to play against.
    - The default number of enemies is set to 0. However, the player can change the number of enemies by one of the following keys. (W, S, up, and down).
    - A cute tank points to the number of enemies that the player has currently selected.
    - The game is stuck in this component until one of the following keys (Enter or space) is pressed.

Figure 2: The enemy choosing screen of our game.
Currently, 0 enemies are selected.

- Game
    - The game consists of two modes. One Player Mode and Two Player Mode.
    - One Player Mode:
        - In this mode, the player can only control one tank, which is the gold tank in the bottom left corner of the gameboard.

        - The enemies' movement is based on the current positions of the player's tank, and it is restricted by the ILLINI walls, the white walls on the edge of the game board, and the other tanks.
        - Each tank can shoot bullets to attack the other tanks. Each bullet costs one life.
        - The bullets' movement is based on the current positions and direction of the tank, and it is restricted by the ILLINI walls, the white walls on the edge of the game board, and the other tanks.
        - The number of hearts correspond to the number of lives that each tank has.
    - Two Player Mode:
        - In this mode, one player can control the tank in the bottom left corner, and the other player can control the tank in the top left corner.
        - Each tank's movement is based on the keyboard key that each player presses, and it is restricted by the ILLINI walls, the white walls on the edge of the game board, and the other tank.
        - Each tank can shoot bullets to attack the other tanks. Each bullet costs one life.
        - The bullets' movement is based on the current positions and direction of the tank, and it is restricted by the ILLINI walls, the white walls on the edge of the game board, and the other tanks.

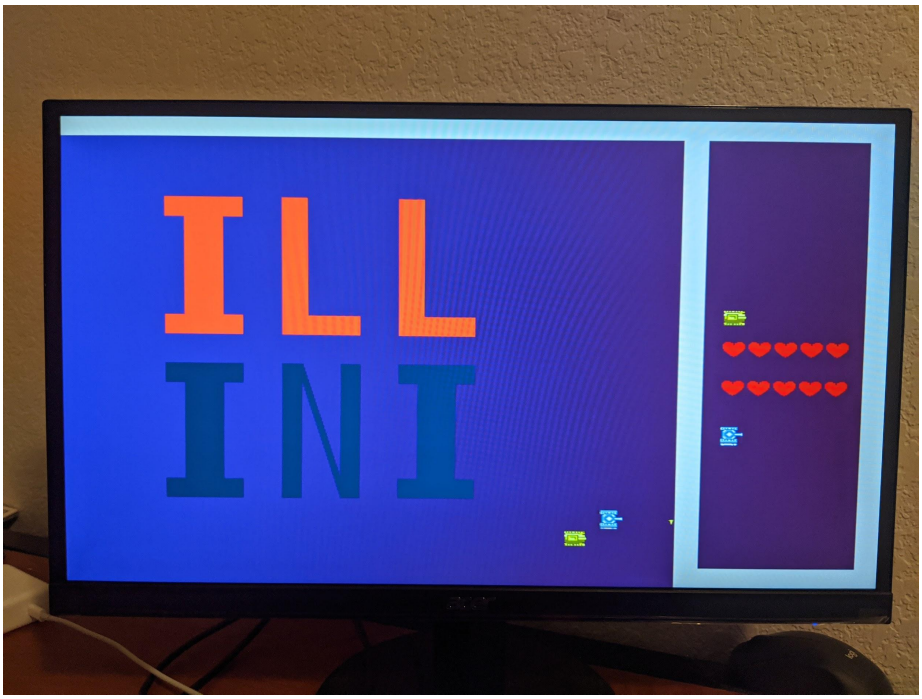


Figure 3.1: The game screen of our game (with one enemy)

Figure 3.2: The game screen of our game (with two enemies)

- End Screen
    - After each round finishes, the game reaches this final component.
    - There are four possible endings in this game:
        - The player **loses** the one player game.
        - The player **wins** the one player game.
        - Player 1 **wins** the two player game.
        - Player 2 **wins** the two player game.
    - In each case, different background images are projected to the screen.
    - There is a bonus scene in our game, which is that if player decide to choose 0 enemy in the one-player-mode choosing enemy stage, then player would automatically win the game and “You Win” would display on the screen,
    - We can then hit Q to return to the start screen.

Figure 4.1: One of four ending screens of our game.
This ending means that the player wins the game in the one-player-mode.

Figure 4.2: One of four ending screens of our game.
This ending means that the player loses the game in the one-player-mode.

Figure 4.3: One of four ending screens of our game.
This ending means that the player1 wins the game in the two-player-mode.

Figure 4.4: One of four ending screens of our game.
This ending means that the player2 wins the game in the two-player-mode.

The exact implementation of each component is further discussed in the next section.

Written Description of Hardware

Hardware-wise, we used the hardware components from lab 8 as a building block to start on our project. The following are groups of hardware additions to make this game work.

Sprite and Background image files

First, we have written a python script to convert the sprite and background images to a txt file (row major) with RGB values that the SystemVerilog compiler can process. Because of the space limitation of the on-chip memory (only 3,981,312 available memory bits, which is only around 3.8 MB), we not only need to shrink the size of the image that we are trying to convert, but we also need to shrink the width of each RGB value. In our particular implementation, we shrink the size of each RGB value from 24 bits (8 bits for R, 8 bits for G, and 8 bits for B) to 12 bits (4 bits for R, 4 bits for G and 4 bits for B) by truncating the least significant bit for each color channel. (For example, if the original R value is 0x89, it will be truncated to 0x8).

Next, we have to load the image txt files to the on-chip memory. We used the RAM sv file developed by Rishi as the building block for our ROMs. However, since we are not planning to directly modify the text file that we just converted (read only), we removed the write address and write data from the sv file.

Finally, after getting the RGB values of each pixel of the image, we need to draw it on the screen. However, since we shrink the size of each RGB value read from the txt file to merely 12 bits, we need to extend the RGB value to 24 bits once the 12-bit data is read. Thus, we designed a palette module, which serves as a lookup table, that translates each 12-bit data to 24-bit data. Then we split the 24-bit data into three 8-bit segments. The first, second, and third segment belongs to the R channel, G channel and B channel. To save some memory, we only included around 40 colors that we absolutely needed.

We not only need to draw the whole image onto the screen, we also need to draw a part of the image (like tanks for example) from the spritesheet at a particular location of the screen. In this case, we designed a separate module called sp_addr_lookup to translate the exact address that we need to read from based on the current x and y coordinates that are being drawn and other game logic signals.

Figure 5: The sprite sheet we utilized while building and designing our final project.

Mode Selection and States

Since our game supports game mode selection, we need to implement the game modes in the top level file of our project. We differentiated different game modes by using a combination of state logic and individual signals:

- Differentiating between one player mode and two player mode:
    - We achieved this task by altering different control signals and creating separate states.
    - If one player mode was chosen, the signal that represents the one player mode becomes high and vice versa.
    - Then the signals are read, if the signal that represents one player mode is high, the next state is routed to the enemy selection state, then routed to the one player game state. If the signal that represents two player mode is high, we skip the enemy selection state and jump to the two player game state.
- Differentiating between the number of enemies
    - We achieved this task by purely manipulating a 4-bit wide control signal.
    - We store the number of enemies chosen into a 4-bit register, which is then read by the top level module.
    - If no enemy is selected, the player automatically wins since the player has no enemy to play against.
    - Otherwise, the enemy is / the enemies are drawn onto the screen.

One more important note: Some of the state transitions is determined by the key pressed on the keyboard from the player. In most occasions, one key press lasts much longer than a clock cycle from the 50 Mhz clock generated from the FPGA board. Problems may arise if, for example, transitions between consecutive states depend on the same keyboard input. In this case, after pressing a key, the state machine will jump several states ahead even though we only mean to go to the next state. To address this situation, we added several wait states after states that require a keyboard input. In our implementation, we jump to a wait state if a particular key is pressed, and we can't leave the wait state until the keycode clears.

Player and Bullet Movement

First, we need to make tanks to move. As mentioned earlier, the movement of player tanks is controlled by the player(s) by pressing keys on the keyboard, if no key is pressed, the tank stays still. In our implementation, each keyboard input is only processed if it meets the rising edge of the frame clock. (60 Mhz). However, because of the limited size of the game board, the tank needs to stop whenever it reaches the border of the game board so that it won't go out of bound.

So we need to stop the motion for a particular direction if going towards that particular direction if the tank will hit the boundary of the game board.

Also, to imitate the true Battle City game experience, the image of the tank to be drawn on the display depends on the current direction of the tank. So, to achieve this, we saved the direction of the tank into a 2-bit register, which is read by the sp_addr_lookup module I described to draw the tank for each of the four directions.

Second, we need the tanks to shoot bullets. Conceptually, it is very straight-forward to implement. It was really similar to the ball.sv we implemented for lab 8, except the bullet should only be shot for a fixed distance after the player prompts. However, the actual implementation proved to be much harder. The first thing we did was we spent time to determine *how* and *when* the bullet should be shot. This might seem to be easy, but it took us a long time to debug since we had a nasty argument with the color mapper about what thing should be drawn onto the screen first. After we resolved this issue, we spent some time determining *where* the bullet should be shot. In our implementation, the direction of the bullet depends on the current direction of the tank that the bullet was shot from. We created several registers to hold the midpoint of the four sides of the tank, the value of each of the four registers representing the start position of the bullet for the tank facing different directions. The registers are updated in each rising edge of the frame clock so that the starting position of the bullet is updated in a timely manner as the player moves their tank. Finally, we need to determine how long (time-wise and distance-wise) the bullet should travel from the tank. In our implementation, we set up a 26 bit counter to count how many clock cycles the bullet should be drawn on the screen. The counter is incremented during the rising edge of the main clock (50 Mhz), if the counter reaches a threshold, we stop drawing the bullet onto the screen and reset the counter.

Third, we need the bullets to stop if it hits an obstacle and create a visual effect. To stop the bullet, we created a signal to stop drawing the bullets if it reaches the edge of an obstacle, regardless of what the counter value is, and automatically reset the counter to prepare for the next bullet. If the bullets hit a wall, we stop drawing the bullet. Instead of letting the bullet simply disappear, we draw a boom image at the stopping point of the bullet for a short amount of time, which is controlled by another 26-bit counter.

Fourth, we need to construct additional walls in the actual game board. This is one of the hardest tasks to implement. We first drew the wall on the gameboard by specifying the color to be drawn on the board for a specific set of DrawX and DrawY. Then, we need to specify the stopping condition for the walls. If the tank falls in range of specific X or Y coordinates, (not at specific X and Y coordinates since the tank might not be able to reach that coordinate thereby unable to stop), we send a signal to stop the tank movement in that particular direction. However, it works

differently for bullets. In our implementation, if the bullet attempts to hit the wall, we send a signal to completely stop the bullet.

Fifth, we need to extend the size of each keycode so we can handle multiple keyboard inputs. On the hardware side, we extended the size of keycode from 8 bits to 32 bits to support 4 simultaneous keyboard inputs. However, not all four keyboard inputs combinations work, this will be discussed in detail in the design decision section later.

Finally, we need to split the section into two subsections to explain the work done for one player mode and two player mode:

In one player mode, we extrapolated what we've learned while designing the player tank to design the enemy tanks. At the first glance, the enemy tanks and the player tanks largely follow the same logic. However, we are not supposed to control the enemy tanks, which makes the task more challenging than it seemed to be. Our First Priority would be designing how the enemy tanks react to the player tanks. Below was the algorithm we designed for the enemy tank movements.

Algorithm:

1. Calculate the direct distance from the enemy tank to the player if the enemy tank goes right, up, left, or down. Choose the direction that minimizes the distance. (Except when it hits an obstacle, then we choose the direction that gives the second least distance to the player tank.)
2. If the enemy tank is close enough to the player tank, we stop the enemy tank and aim the enemy tank to the direction that faces the player tank.

One more important note here is the unlike the player tank controlled by the player, the enemy tank shoots bullets continuously.

Two player mode was much simpler to implement than the one player mode. We only need to add another tank onto the screen, take care of the boundary condition and this is it.

Lives counting and Exit conditions

We used a counter to count for the amount of lives for each tank. The amount of lives for each tank is drawn on the sidebar as the number of hearts. If the opponent tank's bullet hits a tank (if the draw bullet signal and draw signal overlaps), then the number of lives decreases. If either side has run out of lives, we stop the game and display one of the four endings per scenario.

Attempts to add sound to project (failed)

We attempted to add sound to the project by first converting the music to a text file. Then we wrote an audio controller to drive the audio to the audio chip. However, the end result is not very successful. We could hear some beats, however, that was not the sound that we wanted to hear so we stopped working on audio and started working on the final project report instead.

Written Description of Software

For the software part of the final project, we directly utilized the software code from lab8 since the entire game logic was built in hardware. The purpose of our software part was to let the fpga be able to receive the signal from the keyboard. However, little modification has been made on the both software part NIOS II SOC part and so that our keycode could receive 32 bits data from the keyboard, which supports 4 keys pressed at the same time. The reason why we are doing such a thing was that in the two-players-mode, both tanks need to move and shoot at the same time. However, previously designed in the lab8 could only support 1 key pressed, which is not enough for our final project. As a result, we figured out the way with multiple keys.

The modification we have made to the software part compared to lab8 was that in the main function, we changed the data type of the keycode from int to unsigned int. We also need to modify the data type of the keycode base to the volatile unsigned int pointer in order to have the correct pointer pointed to the keycode.

One of the most important modifications we need to make was shown in the picture below.

```
keycode = UsbRead(0x051e);
keycode += (UsbRead(0x0520) << 16);
printf("\nfirst two keycode values are %08x\n",keycode);
// We only need the first keycode, which is at the lower byte of keycode.
// Send the keycode to hardware via PIO.
*keycode_base = keycode;
```

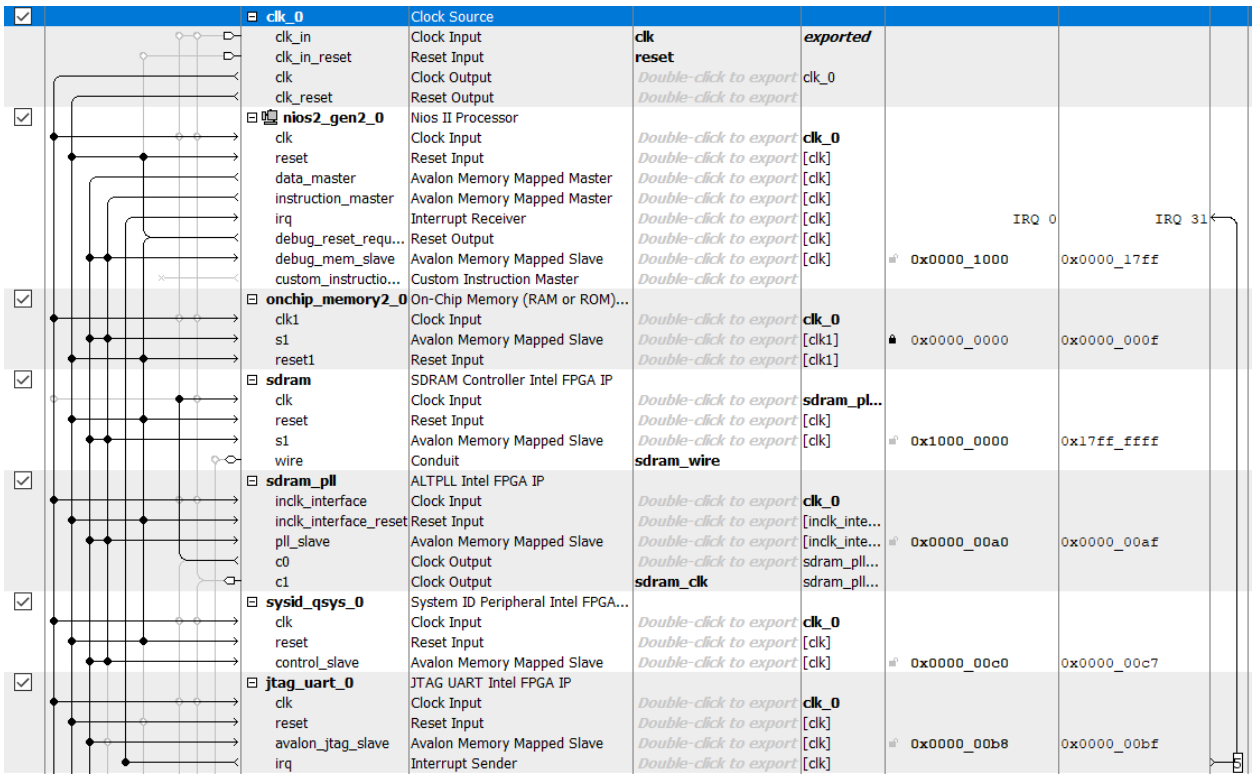
The reason why we did something like this was that the first two keycodes were stored at the address 0x051e in the RAM on the EZ-OTG chip. Each time two keycodes were read from the chip using the UsbRead function because the data width of the EZ-OTG chip is 16 bits. What is more, one PIO block could support up to 4 bytes read, then we just need to modify the code as shown above and our code would be able to support 4 keys pressed at the same time. The rest of the software part would be basically the same with the code in lab8.

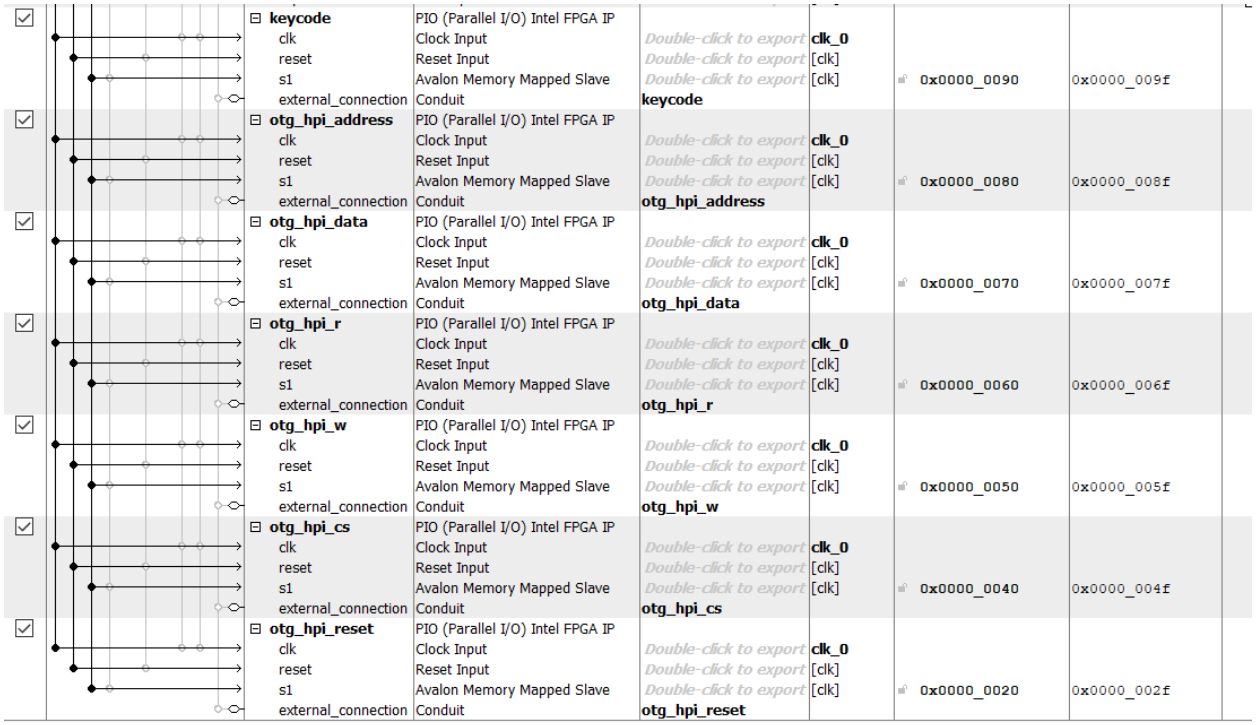
The little modification made to the NIOS II SOC part would be explained in the following section.

Design Decisions

- We drew an ILLINI wall on the game board, because we are proud Illinis!!
- Not all keycode combinations work, sometimes if we press 4 keys together, the keycode will overflow to 01010101, which is not ideal. So, we chose w,a,s,d,c as the keyboard inputs for the first player and 8,u,i,o,h as the keyboard inputs for the second player.

NIOS II SOC description





☑	keycode	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0090	0x0000_009f
	external_connection	Conduit	keycode			
☑	otg_hpi_address	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0080	0x0000_008f
	external_connection	Conduit	otg_hpi_address			
☑	otg_hpi_data	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0070	0x0000_007f
	external_connection	Conduit	otg_hpi_data			
☑	otg_hpi_r	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0060	0x0000_006f
	external_connection	Conduit	otg_hpi_r			
☑	otg_hpi_w	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0050	0x0000_005f
	external_connection	Conduit	otg_hpi_w			
☑	otg_hpi_cs	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0040	0x0000_004f
	external_connection	Conduit	otg_hpi_cs			
☑	otg_hpi_reset	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click to export	clk_0		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0020	0x0000_002f
	external_connection	Conduit	otg_hpi_reset			

(Top-level of the Qsys)

Module: nios2_gen2_0

```
module final_soc_nios2_gen2_0 (
      input  wire        clk,                                //                    clk.clk
      input  wire        reset_n,                            //                  reset.reset_n
      input  wire        reset_req,                          //                        .reset_req
      output wire [28:0] d_address,                          //          data_master.address
      output wire [3:0]  d_byteenable,                       //                        .byteenable
      output wire        d_read,                             //                        .read
      input  wire [31:0] d_readdata,                         //                        .readdata
      input  wire        d_waitrequest,                      //                        .waitrequest
      output wire        d_write,                            //                        .write
      output wire [31:0] d_writedata,                        //                        .writedata
      output wire        debug_mem_slave_debugaccess_to_roms, //                        .debugaccess
      output wire [28:0] i_address,                          //   instruction_master.address
      output wire        i_read,                             //                        .read
      input  wire [31:0] i_readdata,                         //                        .readdata
      input  wire        i_waitrequest,                      //                        .waitrequest
      input  wire [31:0] irq,                                //                 irq.irq
      output wire        debug_reset_request,                //  debug_reset_request.reset
      input  wire [8:0]  debug_mem_slave_address,            //      debug_mem_slave.address
      input  wire [3:0]  debug_mem_slave_byteenable,         //                        .byteenable
      input  wire        debug_mem_slave_debugaccess,        //                        .debugaccess
      input  wire        debug_mem_slave_read,               //                        .read
      output wire [31:0] debug_mem_slave_readdata,           //                        .readdata
      output wire        debug_mem_slave_waitrequest,        //                        .waitrequest
      input  wire        debug_mem_slave_write,              //                        .write
      input  wire [31:0] debug_mem_slave_writedata,          //                        .writedata
      output wire        dummy_ci_port                       // custom_instruction_master.readra
  );
```

Description: The economic version of processor we utilized for the NIOS II, which could handle the operations and computations our program utilized.

Module: onchip_memory2_0

```
module final_soc_onchip_memory2_0 (
                                    // inputs:
                                     address,
                                     byteenable,
                                     chipselect,
                                     clk,
                                     clken,
                                     freeze,
                                     reset,
                                     reset_req,
                                     write,
                                     writedata,

                                    // outputs:
                                     readdata
                                  )
;
```

Description: The onchip_memory module is often used as a cache or data buffer for our system and it is tightly coupled to FPGA logic. The speed of the on-chip-memory is relatively fast and we could directly instantiate it using system verilog. However, it only has 3.888 Mbits.

Module: sdram

```
module final_soc_sdram_input_efifo_module (
                                            // inputs:
                                             clk,
                                             rd,
                                             reset_n,
                                             wr,
                                             wr_data,

                                            // outputs:
                                             almost_empty,
                                             almost_full,
                                             empty,
                                             full,
                                             rd_data
                                          )
;
```

Description: This is the sdram controller for the NIOS II CPU, which needs to be refreshed every time we do operations. The size of the sdram is 16 Mbits, which is larger than that of the on-chip-memory. However, we would need to use the control panel to put data into the sdram and the speed of the sdram is a little slower than that of the on-chip-memory.
Exports: sdram_wire

Module: sdram_pll

```
module  final_soc_sdram_pll
   ( address,  areset,   c0,
     c1,       clk,     configupdate,
   locked,  phasecounterselect,     phasedone,
   phasestep,  phaseupdown,   read,
   readdata,
   reset,
   scanclk,
   scanclkena,
   scandata,
   scandataout,
   scandone,
   write,
   writedata) /* synthesis synthesis_clearbox=1 */;
   input   [1:0]  address;
   input   areset;
   output   c0;
   output   c1;
   input   clk;
   input   configupdate;
   output   locked;
   input   [3:0]  phasecounterselect;
   output   phasedone;
   input   phasestep;
   input   phaseupdown;
   input   read;
   output   [31:0]  readdata;
   input   reset;
   input   scanclk;
   input   scanclkena;
   input   scandata;
   output   scandataout;
   output   scandone;
   input   write;
   input   [31:0]  writedata;
```

Description: This module is the pll for the system. This block could produce a clock. However, this clock is able to produce a clock with a 3ns phase shift prior to the main clock. The reason why we would need this is because sdram would have the ability to work 3ns before the main clock. As a result, the data to SDRAM would be stabilized before feeding itself to the SDRAM.
Exports: sdram_clk

Module: sysid_qsys_0

```
module final_soc_sysid_qsys_0 (
                // inputs:
                 address,
                 clock,
                 reset_n,

                // outputs:
                 readdata
              )
;
```

Description: This is the system ID checker, which makes sure that software could be compatible with the hardware.

Module: jtag_uart_0

```
module final_soc_jtag_uart_0 (
                               // inputs:
                                av_address,
                                av_chipselect,
                                av_read_n,
                                av_write_n,
                                av_writedata,
                                clk,
                                rst_n,

                               // outputs:
                                av_irq,
                                av_readdata,
                                av_waitrequest,
                                dataavailable,
                                readyfordata
                             )
```

Description: The module here could let the fpga board communicate with the PC so that the scanf and printf function in C would be able to work. We could send the data and read from and to the fpga board.

Module: keycode

```
module final_soc_keycode (
                           // inputs:
                            address,
                            chipselect,
                            clk,
                            reset_n,
                            write_n,
                            writedata,

                           // outputs:
                            out_port,
                            readdata
                         )
;
```

Description: The PIO block we have created in order to transmit the key we pressed on the keyboard in hex. However, instead of 8 bits, I extended the bits of the keycode to 32 bits due to changes made in the software parts. As a result, the keycode now could support reading of 4 key codes simultaneously.

Exports: keycode

Module: otg_hpi_address

```
module final_soc_otg_hpi_address (
                                   // inputs:
                                    address,
                                    chipselect,
                                    clk,
                                    reset_n,
                                    write_n,
                                    writedata,

                                   // outputs:
                                    out_port,
                                    readdata
                                 )
;

  output  [  1: 0] out_port;
  output  [ 31: 0] readdata;
  input   [  1: 0] address;
  input            chipselect;
  input            clk;
  input            reset_n;
  input            write_n;
  input   [ 31: 0] writedata;
```

Description: This is a 2-bit signal representing the reading address of the ETZ_OTG chip. The detailed way of accessing the chip is shown in the picture below.

Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

Exports: otg_hpi_address

Module: otg_hpi_data

```
module final_soc_otg_hpi_data (
                                // inputs:
                                 address,
                                 chipselect,
                                 clk,
                                 in_port,
                                 reset_n,
                                 write_n,
                                 writedata,

                                // outputs:
                                 out_port,
                                 readdata
                              )
;

  output  [ 15: 0] out_port;
  output  [ 31: 0] readdata;
  input   [  1: 0] address;
  input            chipselect;
  input            clk;
  input   [ 15: 0] in_port;
  input            reset_n;
  input            write_n;
  input   [ 31: 0] writedata;
```

Description: This is a 16-bit inout signal representing the data read or write from or to ETZ-OTG chip.
Exports: otg_hpi_data

Module: otg_hpi_r
Description: This is a 1 bit output data, which enables it to read from the ETZ-OTG chip.
Exports: otg_hpi_r

Module: otg_hpi_w
Description: This is a 1 bit output data, which enables it to write to the ETZ-OTG chip.
Exports: otg_hpi_w

Module: otg_hpi_cs

```
module final_soc_otg_hpi_cs (
                               // inputs:
                                address,
                                chipselect,
                                clk,
                                reset_n,
                                write_n,
                                writedata,

                               // outputs:
                                out_port,
                                readdata
                             )
;
```

Description: This is a 1 bit output data, which is the chip select signal on the ETZ-OTG chip.
Exports: otg_hpi_cs

Module: otg_hpi_reset
Description: This is a 1 bit output data, which enables it to reset the ETZ-OTG chip.
Exports: otg_hpi_reset

Module Descriptions

Lab8 in lab8.sv

```
module lab8( input              CLOCK_50,
             input        [3:0] KEY,          //bit 0 is set up as Reset
             output logic [6:0] HEX0, HEX1, HEX2, HEX3,
             // VGA Interface
             output logic [7:0] VGA_R,        //VGA Red
                                VGA_G,        //VGA Green
                                VGA_B,        //VGA Blue
             output logic       VGA_CLK,      //VGA Clock
                                VGA_SYNC_N,   //VGA Sync signal
                                VGA_BLANK_N,  //VGA Blank signal
                                VGA_VS,       //VGA virtical sync signal
                                VGA_HS,       //VGA horizontal sync signal
             // CY7C67200 Interface
             inout  wire  [15:0] OTG_DATA,    //CY7C67200 Data bus 16 Bits
             output logic [1:0] OTG_ADDR,     //CY7C67200 Address 2 Bits
             output logic       OTG_CS_N,     //CY7C67200 Chip Select
                                OTG_RD_N,     //CY7C67200 Write
                                OTG_WR_N,     //CY7C67200 Read
                                OTG_RST_N,    //CY7C67200 Reset
             input              OTG_INT,      //CY7C67200 Interrupt
             // SDRAM Interface for Nios II Software
             output logic [12:0] DRAM_ADDR,   //SDRAM Address 13 Bits
             inout  wire  [31:0] DRAM_DQ,     //SDRAM Data 32 Bits
             output logic [1:0] DRAM_BA,      //SDRAM Bank Address 2 Bits
             output logic [3:0] DRAM_DQM,     //SDRAM Data Mast 4 Bits
             output logic       DRAM_RAS_N,   //SDRAM Row Address Strobe
                                DRAM_CAS_N,   //SDRAM Column Address Strobe
                                DRAM_CKE,     //SDRAM Clock Enable
                                DRAM_WE_N,    //SDRAM Write Enable
                                DRAM_CS_N,    //SDRAM Chip Select
                                DRAM_CLK      //SDRAM Clock
                 );
```

- **Purpose:** This module is the top-level module of our project. Before you ask anything, yes, we did not bother to change the name.
- **Description**: This module instantiates all submodules needed for the final project.

Enemy1 in enemy.sv

```
module enemy1 (input Clk, Reset, frame_clk,
                   input [9:0] DrawX, DrawY,
                   input [9:0] playerX, playerY,
                   input enable1, enable12,
                   input [9:0] Enemy2_X_Pos_out, Enemy2_Y_Pos_out,
                   output [9:0] Right_mid_X_en1, Right_mid_Y_en1, Up_mid_X_en1, Up_mid_Y_en1, Left_mid_X_en1,
                                Left_mid_Y_en1, Down_mid_X_en1, Down_mid_Y_en1,
                   output [9:0] Enemy1_X_Pos_out, Enemy1_Y_Pos_out,
                   output [19:0] Enemy1_DisX, Enemy1_DisY,
                   output is_enemy1,
                   output [1:0] is_enemy1_out,
                   output [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                   output is_bullet, is_boom,
                   output [19:0] Dist_X_boom_en1, Dist_Y_boom_en1,
                   output player_hit
                  );
```

- **Purpose**: This module is the module for the first enemy in the one player mode.
- **Description**: This module takes the current mode, player position, etc as input and it generates the position and the direction of the enemy to be drawn on the screen.

Enemy2 in enemy.sv

```
module enemy2 (input Clk, Reset, frame_clk,
                   input [9:0] DrawX, DrawY,
                   input [9:0] playerX, playerY,
                   input enable12,
                   input enable1,
                   input [9:0] Enemy1_X_Pos_out, Enemy1_Y_Pos_out,
                   output [9:0] Right_mid_X_en2, Right_mid_Y_en2, Up_mid_X_en2, Up_mid_Y_en2,
                                Left_mid_X_en2, Left_mid_Y_en2, Down_mid_X_en2, Down_mid_Y_en2,
                   output [9:0] Enemy2_X_Pos_out, Enemy2_Y_Pos_out,
                   output [19:0] Enemy2_DisX, Enemy2_DisY,
                   output is_enemy2,
                   output [1:0] is_enemy2_out,
                   output [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                   output is_bullet, is_boom,
                   output [19:0] Dist_X_boom_en2, Dist_Y_boom_en2,
                   output player_hit
                  );
```

- **Purpose** and **Description** similar to enemy1.

Tanks_selector in selector.sv

```
module tanks_selector(
                        input [31:0] keycode,
                        input VGA_CLK,
                        input[9:0] DrawX, DrawY,
                        output are_two_tanks,
                        output is_one_tank,
                        output one_player_mode,
                        output two_players_mode,
                        input Reset
                      );
```

- **Purpose:** This module is used to select the game mode on the start screen.
- **Description:** This module draws a small tank next to the currently selected game mode ,which can be changed by keypresses.

en_draw in selector.sv

```
module en_draw (input [31:0] keycode,
                input VGA_CLK, Reset, en_state,
                input [9:0] DrawX, DrawY,
                output is_tank, is_count, is_number, chosen, is_ennumber,
                output[3:0] en_num
                );
```

- **Purpose**: This module is used to select the enemy to play against for one player mode.
- **Description**: This module draws a small tank next to the currently selected number of enemies, which can be changed by keypresses.

en_game in selector.sv

```
module en_game (input [31:0] keycode,
                input VGA_CLK, Reset, game_mode,
                input [9:0] DrawX, DrawY,
                input [3:0] en_num,
                output is_player1, is_player2, is_player_en2
                );
```

- **Purpose**: This module is used to draw the tanks under/over the health bar to the right of the game board.
- **Description**: This module generates the signal to draw to the tanks under/over the health bar to the right of the game board based on current mode and number of enemies.

Sp_addr_lookup in sp_addr_lookup.sv

```
module sp_addr_lookup(
                    input is_one_tank, are_two_tanks,
                          is_tank_final, is_tank_final2,
                          is_enemy1_final, is_enemy2_final,
                          en_tank, en_count, en_number,
                          is_ennumber, is_boom, is_boom2,
                          is_boom_en1, is_boom_en2,
                          is_player1, is_player2, one_player_mode,
                          is_player_en2,
                    input [1:0] is_tank_out, is_tank_out2, is_enemy1_out, is_enemy2_out,
                    input [1:0] mode,
                    input [5:0] player_lives, enemy1_lives, enemy2_lives,
                    input [5:0] player1_lives, player2_lives,
                    input [9:0] DrawX, DrawY,
                    input [19:0] DistX, DistY, DistX2, DistY2,
                    input [19:0] Enemy1_DisX, Enemy1_DisY,
                    input [19:0] Enemy2_DisX, Enemy2_DisY,
                    input [19:0] Dist_X_boom, Dist_Y_boom,
                    input [19:0] Dist_X_boom2, Dist_Y_boom2,
                    input [19:0] Dist_X_boom_en1, Dist_Y_boom_en1,
                    input [19:0] Dist_X_boom_en2, Dist_Y_boom_en2,
                    input [3:0] en_num,
                    output [19:0] Addr_out
                );
```

- **Purpose**: This module generates the sprite sheet address to read data.
- **Description**: The sprite sheet address is generated by taking account of the control signals, state encodings and the game modes.

spROM in spROM.sv

```
module spROM (
                    input [19:0] read_address,
                    input Clk,
                    output logic [11:0] data_Out
                );
```

- **Purpose & Description**: This module reads data from a sprite sheet address.

Tank2 in tank2.sv

```
module  tank2( input         Clk,                // 50 MHz clock
                             Reset,              // Active-high reset signal
                             frame_clk,          // The clock indicating a new frame (~60Hz)
                                    two_players_mode,   // Two_players_mode
               input  [31:0] keycode,
             input  [9:0]  DrawX, DrawY,       // Current pixel coordinates
               input  stop_tank_right, stop_tank_up, stop_tank_left, stop_tank_down,
               output [9:0]  Right_mid_X, Right_mid_Y,
              output [9:0]  Up_mid_X, Up_mid_Y,
               output [9:0]  Left_mid_X, Left_mid_Y,
               output [9:0]  Down_mid_X, Down_mid_Y,
               output [19:0] DistX_out, DistY_out,
               output [9:0]  tank_X_Pos_out, tank_Y_Pos_out,
             output logic  is_tank,             // Whether current pixel belongs to tank or background
               output [1:0]  is_tank_out
            );
```

- **Purpose:** This is the module for the second tank in two player mode.
- **Description:** This module generates the direction and location coordinates based on the keyboard inputs and other control signals.

VGA_controller in VGA_controller.sv

```
module  VGA_controller (input              Clk,
                                           Reset,
                        output logic       VGA_HS,
                                           VGA_VS,
                        input              VGA_CLK,
                        output logic       VGA_BLANK_N,
                                           VGA_SYNC_N,

                        output logic [9:0] DrawX,
                                           DrawY
                        );
```

- **Description**: This module generates how the pixels should be drawn by generating the horizontal, the vertical pulse and the blanking interval.
- **Purpose**: This module serves a hardware for the corresponding VGA protocol.

Wall2 in wall2.sv

```
module wall2( input[9:0] DrawX, DrawY,
              input[9:0] Ball_X_Pos, Ball_Y_Pos,
              input[9:0] Bullet_X_Pos, Bullet_Y_Pos,
              input [9:0] bulletx1, bullety1, ballx, bally,
              input enable2,
              input draw_bullet, draw_boom,
              output stop_tank_right, stop_tank_up, stop_tank_left, stop_tank_down,
              output stop_bullet,
              output stop_bullet21, player1_hit
             );
```

- **Description:** This module contains a list of obstacles for the second tank.
- **Purpose:** This module generates a list of control signals whenever it hits an obstacle, a wall, or another tank.

Wall in wall.sv

```
module wall( input[9:0] DrawX, DrawY,
             input[9:0] Ball_X_Pos, Ball_Y_Pos,
             input[9:0] Bullet_X_Pos, Bullet_Y_Pos,
             input[9:0] enemy1_X_Pos, enemy1_Y_Pos,
             input[9:0] enemy2_X_Pos, enemy2_Y_Pos,
             input[9:0] bulletx2, bullety2, ballx2, bally2,
             input      enable1, enable2, enable12,
             output draw_wall, draw_wall_o, draw_wall_b,
             input draw_bullet, draw_boom,
             output stop_tank_right, stop_tank_up, stop_tank_left, stop_tank_down,
             output stop_bullet, player_hit_en1, player_hit_en2,
             output stop_bullet2, player2_hit
//           output stop_tank_right2, stop_tank_up2, stop_tank_left2, stop_tank_down2,
            );
```

- **Description & Purpose**: Similar to the previous module except this module is for the first tank.

Hpi_io_intf in hpi_io_intf.sv

```
module hpi_io_intf( input        Clk, Reset,
                    input [1:0]  from_sw_address,
                    output[15:0] from_sw_data_in,
                    input [15:0] from_sw_data_out,
                    input        from_sw_r, from_sw_w, from_sw_cs, from_sw_reset, // Ac
                    inout [15:0] OTG_DATA,
                    output[1:0]  OTG_ADDR,
                    output       OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N // Active low
                   );
```

- **Purpose**: This is the interface between NIOS II and EZ-OTG chips.

HexDriver in HexDriver.sv

```
module HexDriver (input  [3:0]  In0,
                  output logic [6:0]  Out0);
```

- **Description**: This is a 16-to-1 MUX selected with the value of the input signal `In0`.
- **Purpose**: This module is used to generate the hex number (that corresponds to the value of keystroke) on the FPGA board.

heart_mapper_2p in heart_2p.sv

```
module heart_mapper_2p(
                    input Clk,
                    input heart_enable,
                    input [9:0] DrawX, DrawY,
                    input [5:0] remaining_hearts,
                    input [5:0] remaining_hearts_en,
                    output is_heart,
                    output [11:0] code
                );
```

- **Purpose**: This module is responsible for the health bar to the right of the game board.
- **Description**: This module outputs the rgb value and a control signal based on DrawX, DrawY and the number of remaining lives.

heartRom in heart.sv

```
module heartRom(
                    input [8:0] read_address,
                    input Clk,
                    output [11:0] data_Out
                );
```

- **Purpose & Description**: This module read data from a heartROM address.

heart_mapper in heart.sv

```
module heart_mapper(
                    input Clk,
                    input heart_enable,
                    input [9:0] DrawX, DrawY,
                    input [5:0] remaining_hearts,
                    input [5:0] remaining_hearts_en,
                    output is_heart,
                    output [11:0] code
                );
```

- **Purpose & Description**: Similar to heart_mapper_2p above but this module is for one player mode with one enemy.

Heart_mapper1 in heart.sv

```
module heart_mapper1(
                    input Clk,
                    input heart_enable, enable12,
                    input [9:0] DrawX, DrawY,
                    input [5:0] remaining_hearts,
                    input [5:0] remaining_hearts_en,
                    input [5:0] remaining_hearts_en2,
                    output is_heart,
                    output [11:0] code
                );
```

- **Purpose & Description**: Similar to heart_mapper_2p above but this module is for one player mode with two enemies.

enemy_wall in enemy_wall.sv

```
module enemy_wall( input [9:0] enemyX, enemyY,
                   input [9:0] playerX, playerY,
                  input [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                   input enable, draw_boom, draw_bullet,
                   input enable_other,
                   input [9:0] otherX, otherY,
                   output stop_enemy_right, stop_enemy_up,
                       stop_enemy_left, stop_enemy_down,
                   output stop_bullet,
                   output player_hit
                  );
```

- **Purpose & Description**: Similar to the wall module above but this module is for the first enemy of one player mode.

enemy_wall2 in enemy_wall.sv

```
module enemy_wall2( input [9:0] enemyX, enemyY,
                    input [9:0] playerX, playerY,
                  input [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                    input enable, draw_boom, draw_bullet,
                    input enable_other,
                    input [9:0] otherX, otherY,
                    output stop_enemy_right, stop_enemy_up,
                        stop_enemy_left, stop_enemy_down,
                    output stop_bullet,
                    output player_hit
                   );
```

- **Purpose & Description**: Similar to the enemy_wall module above but this module is for the second enemy of one player mode.

color_mapper in color_mapper.sv

```
module  color_mapper ( input                                  // Whether current pixel belong
                                                              //   or bac
                                                    is_one_tank,        // Single_Player_
                                                    are_two_tanks,      // Multiplayer Ma
                                                    is_tank_final,
                                                    is_bullet, is_bullet2,
                                                    is_boom, is_boom2,
                                                    draw_wall, draw_wall_o, draw_wall_b,
                                                    is_tank_final2, is_enemy1_final,
                                                    is_bullet_en1, is_boom_en1,
                                                    is_bullet_en2, is_boom_en2,
                       input        [9:0] DrawX, DrawY,        // Current pixel coordinates
                             input             [11:0] code,              // Current color c
                       output logic [7:0] VGA_R, VGA_G, VGA_B, // VGA RGB output
                             input        [1:0] mode
                     );
```

- **Purpose**: This module decides which color should be drawn at each pixel.
- **Description**: This module determines which color should be drawn at each pixel by the game mode and other control signals.

palette in palette.sv

```
module palette (
                    input logic [11:0] RGB_12,
                    output logic [7:0] R, G, B,
                        input logic [9:0] DrawX
                    );
```

- **Purpose & Description**: This module translates the 12-bit RGB value read from the txt file into a standard 24-bit RGB value. (8-bit each).

Tank1 in ball.sv

```
module  tank1( input         Clk,                // 50 MHz clock
                             Reset,              // Active-high reset signal
                             frame_clk,          // The clock indicating a new frame (~60H
               input [31:0]   keycode,
              input [9:0]   DrawX, DrawY,       // Current pixel coordinates
               input stop_tank_right, stop_tank_up, stop_tank_left, stop_tank_down,
               output [9:0]  Right_mid_X, Right_mid_Y,
              output [9:0]  Up_mid_X, Up_mid_Y,
               output [9:0]  Left_mid_X, Left_mid_Y,
               output [9:0]  Down_mid_X, Down_mid_Y,
               output [19:0] DistX_out, DistY_out,
               output [9:0] Ball_X_Pos_out, Ball_Y_Pos_out,
              output logic  is_tank,             // Whether current pixel belongs to ball
               output [1:0] is_tank_out           //direction of the tank
             );
```

- Purpose & Description: Similar to the tank2 module, this module is responsible for the first player in both game modes.

Bullets1 in bullets.sv

```
module bullets1(input [1:0] is_tank_in,
                input Clk,
                input frame_clk,
                input Reset,
                input [31:0] keycode,
                input  [9:0] DrawX, DrawY,
                input [9:0]  Right_mid_X, Right_mid_Y,
                input [9:0]  Up_mid_X, Up_mid_Y,
                input [9:0]  Left_mid_X, Left_mid_Y,
                input [9:0]  Down_mid_X, Down_mid_Y,
                input [1:0] mode,
                input draw_wall_o, draw_wall_b,
                input stop_bullet,
                output [19:0] Dist_X_boom, Dist_Y_boom,
                output [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                output is_bullet, is_boom,
                output draw_bullet, draw_boom,
                output [1:0] is_bullet_out
                );
```

- **Purpose:** This is the module that is responsible for the bullet movement of the first tank.
- **Description:** This module output the current position and direction of the bullet shot from the first tank.

Bullets2 in bullets2.sv

```
module bullets2(input [1:0] is_tank_in,
                input Clk,
                input frame_clk,
                input Reset,
               input is_tank_final2,
                input [31:0] keycode,
                input [9:0] DrawX, DrawY,
                input [9:0]  Right_mid_X, Right_mid_Y,
                input [9:0]  Up_mid_X, Up_mid_Y,
                input [9:0]  Left_mid_X, Left_mid_Y,
                input [9:0]  Down_mid_X, Down_mid_Y,
                input two_players_mode,
                input stop_bullet,
                output [19:0] Dist_X_boom, Dist_Y_boom,
                output [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                output is_bullet, is_boom,
                output draw_bullet, draw_boom,
                output [1:0] is_bullet_out
                );
```

- **Purpose & Description:** Similar to the previous module, this module is designed for the bullet movement of the second tank.

Enemy_bullets in enemy_wall.sv

```
module enemy_bullets (
                        input [1:0] is_enemy_in,
                        input Clk, frame_clk, Reset,
                        input [9:0] DrawX, DrawY,
                        input [9:0]  Right_mid_X, Right_mid_Y,
                        input [9:0]  Up_mid_X, Up_mid_Y,
                        input [9:0]  Left_mid_X, Left_mid_Y,
                      input [9:0]  Down_mid_X, Down_mid_Y,
                        output [9:0] Bullet_X_Pos, Bullet_Y_Pos,
                        output is_bullet, is_boom,
                        output [1:0] is_bullet_out,
                        output [19:0] Dist_X_boom, Dist_Y_boom,
                        input stop_bullet, enable,
                        output draw_boom, draw_bullet
                       );
```

- **Purpose & Description:** Similar to the previous modules, this module is designed for the bullet movements from both enemy tanks.

bgROM in backgroundROM.sv

```
module bgROM

(
    input [19:0] read_address,
    input Clk,
    output logic [11:0] data_Out
);
```

- **Purpose & Description:** This module reads data from a background address.

ennumROM in backgroundROM.sv

```
module ennumROM
(
        input [19:0] read_address,
        input Clk,
        output logic [11:0] data_Out
);
```

- Purpose & Description: This module reads data from an enemy image address.

endROM in endROM.sv

```
module endROM
(
        input [19:0] read_address,
        input Clk,
        output logic [11:0] data_Out
);
```

- Purpose & Description: This module reads data from an end screen image address.

audioRAM in audioRAM.sv

```
module  audioRAM
(
        input [3:0] data_In,
        input [18:0] write_address, read_address,
        input we, Clk,

        output logic [15:0] data_Out
);
```

- **Purpose & Description**: This module reads data from an audio text address.

audio_controller in audio_controller.sv

```
module audio_controller(
                      input logic Clk, Reset,
                      input logic AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK,
                      output logic AUD_DACDAT, AUD_XCK, I2C_SCLK, I2C_SDAT
);
```

- **Purpose & Description**: This module serves as the audio controller for this final project.

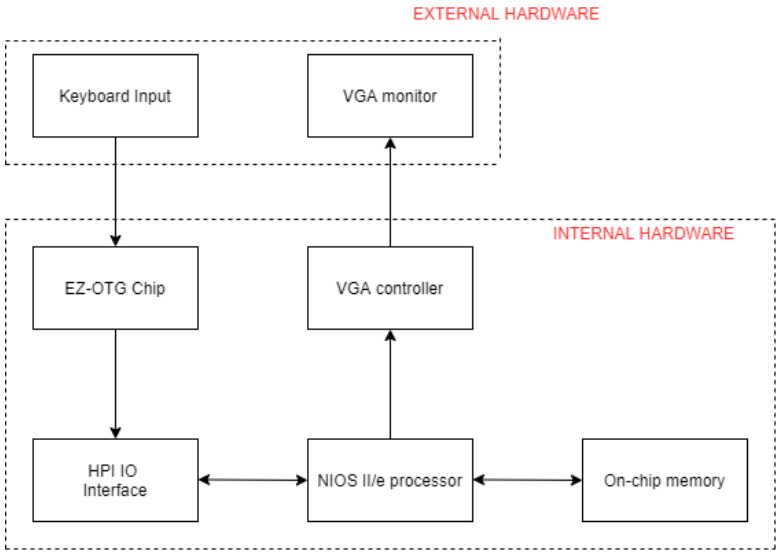
audio_interface.vhd

```
ENTITY audio_interface IS
   PORT
   (
      LDATA, RDATA            :       IN  std_logic_vector(15 downto 0); -- parallel external data inputs
      clk, Reset, INIT        :       IN  std_logic;
      INIT_FINISH             :       OUT std_logic;
      adc_full                :       OUT std_logic;
      data_over               :       OUT std_logic; -- sample sync pulse
      AUD_MCLK                :       OUT std_logic; -- Codec master clock OUTPUT
      AUD_BCLK                :       IN  std_logic; -- Digital Audio bit clock
      AUD_ADCDAT              :       IN  std_logic;
      AUD_DACDAT              :       OUT std_logic; -- DAC data line
      AUD_DACLRCK, AUD_ADCLRCK:       IN  std_logic; -- DAC data left/right select
      I2C_SDAT                :       OUT std_logic; -- serial interface data line
      I2C_SCLK                :       OUT std_logic;  -- serial interface clock
      ADCDATA                 :       OUT std_logic_vector(31 downto 0)
   );
END audio_interface;
```

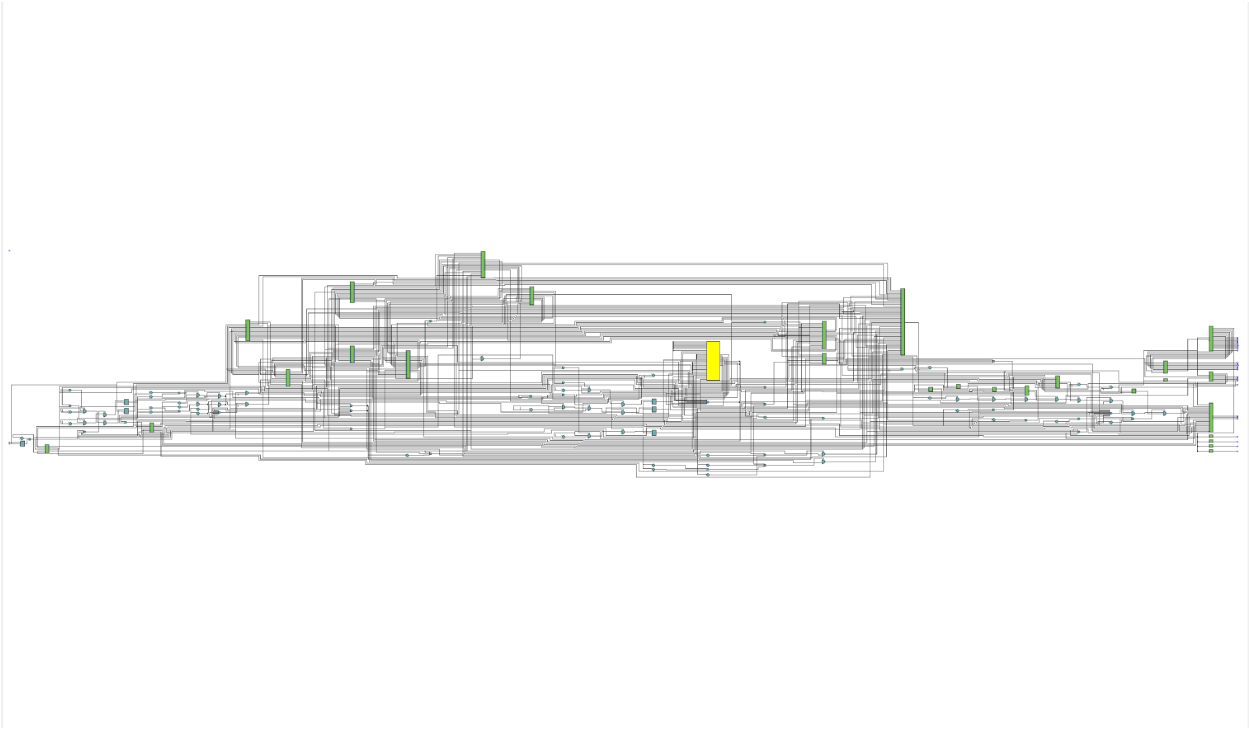
- **Purpose & Description**: This module is the given interface for the audio driver.

Block Diagrams of project & select modules

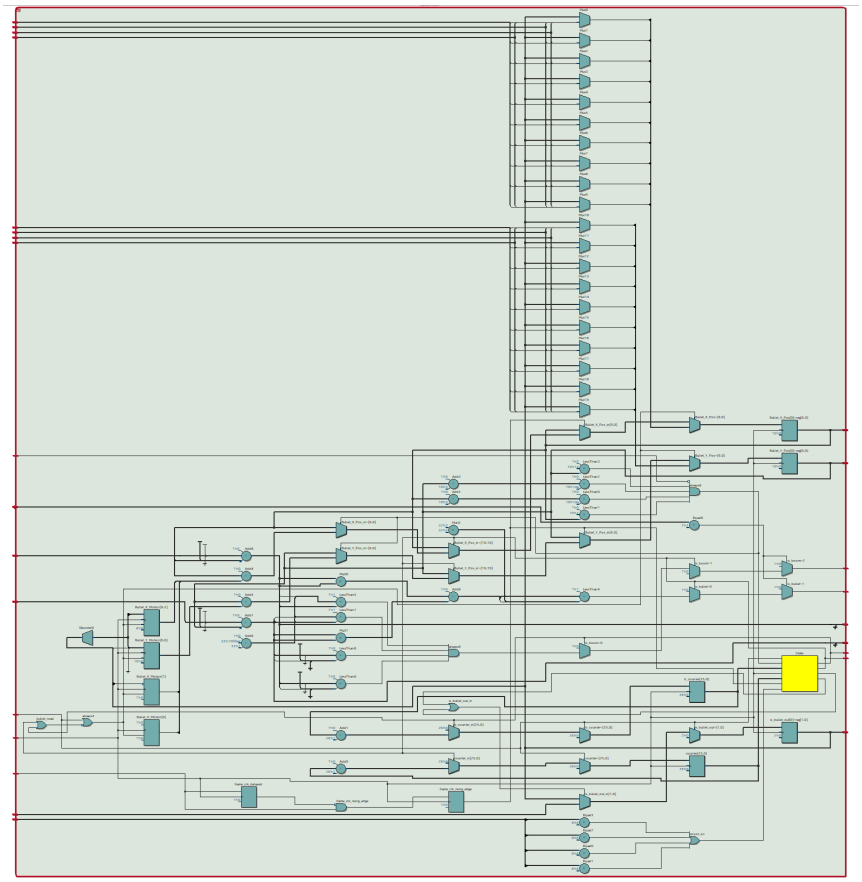
Simplified Block Diagram of Top Level Project



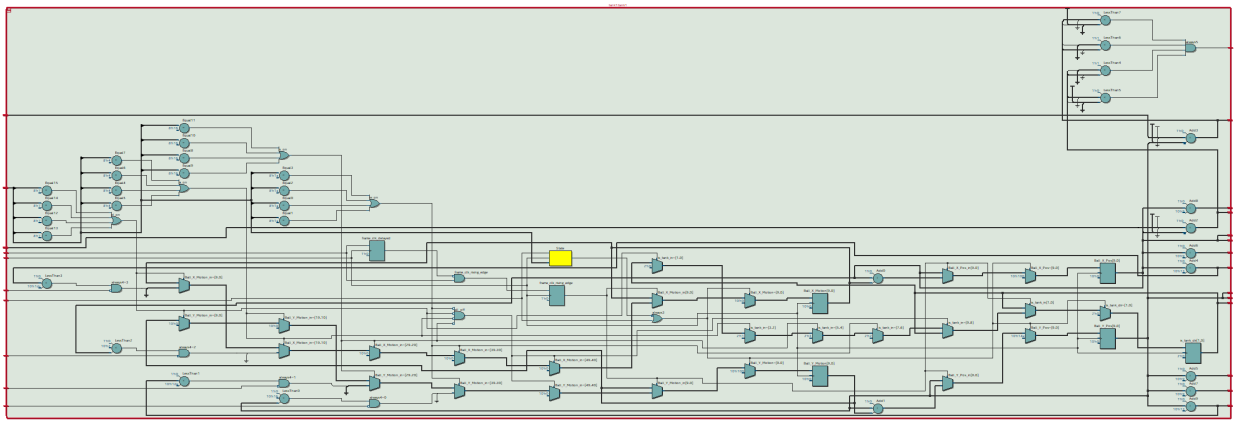
RTL view of the whole project



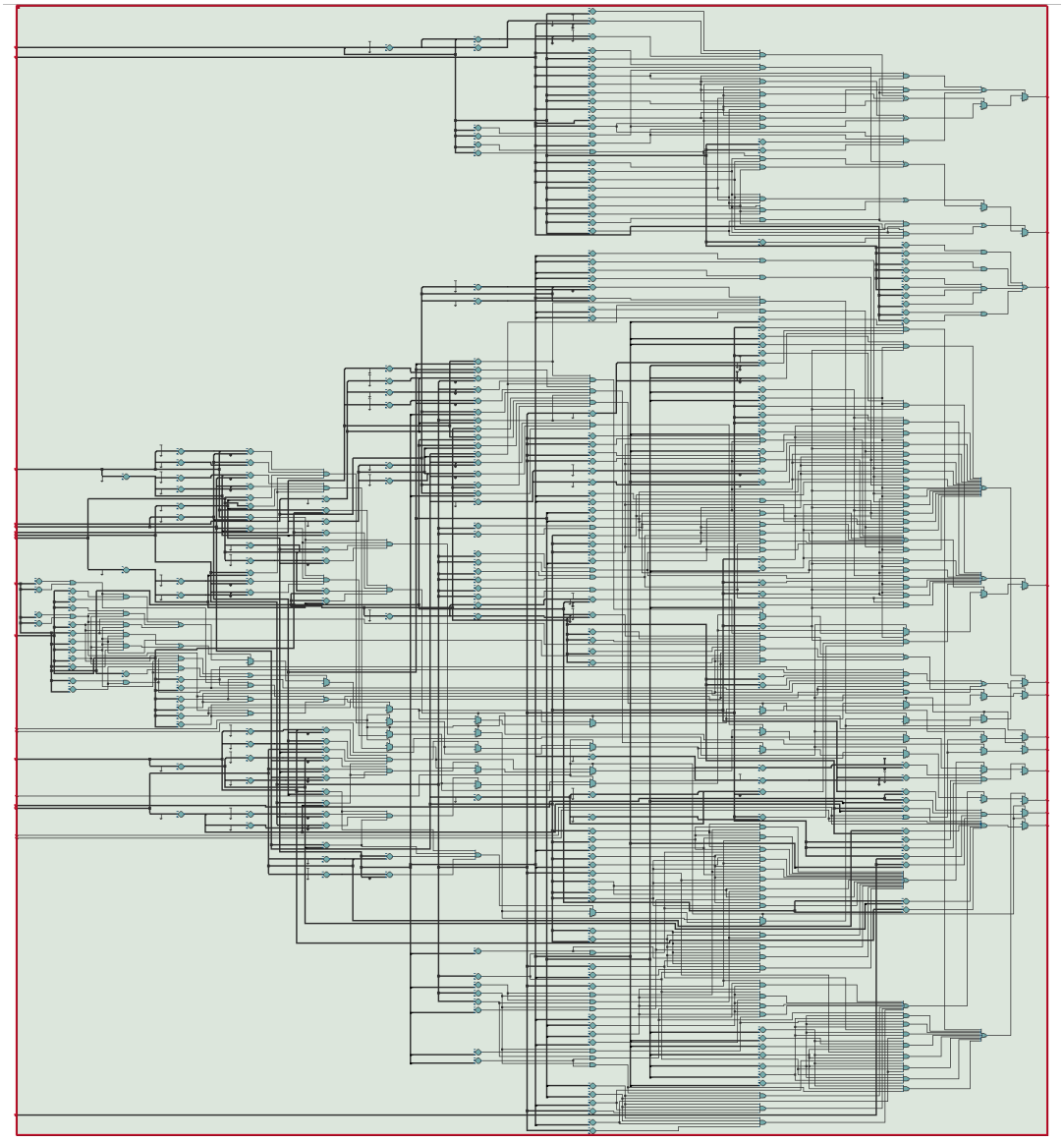
RTL view of module bullets1



RTL view of module tank1

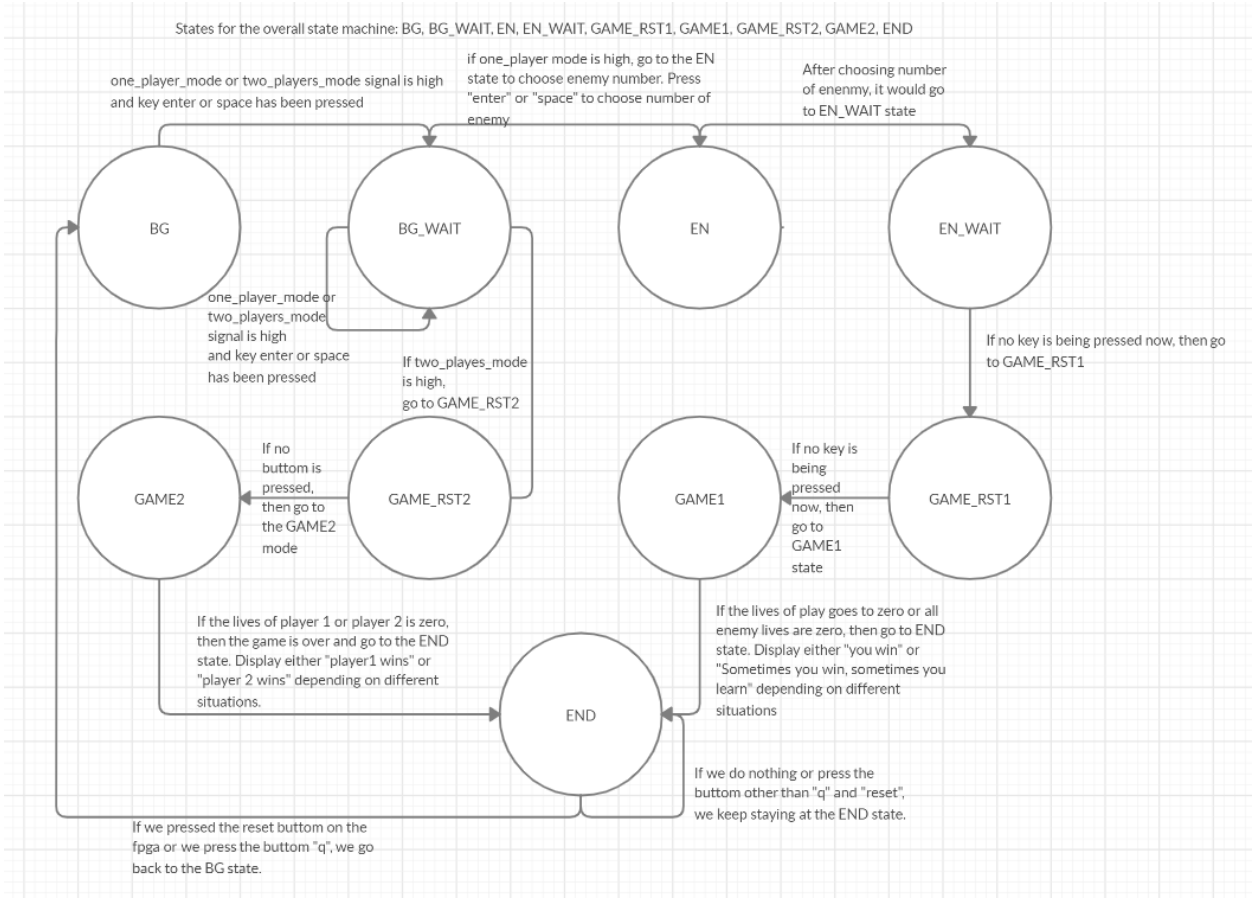


RTL view of module wall1



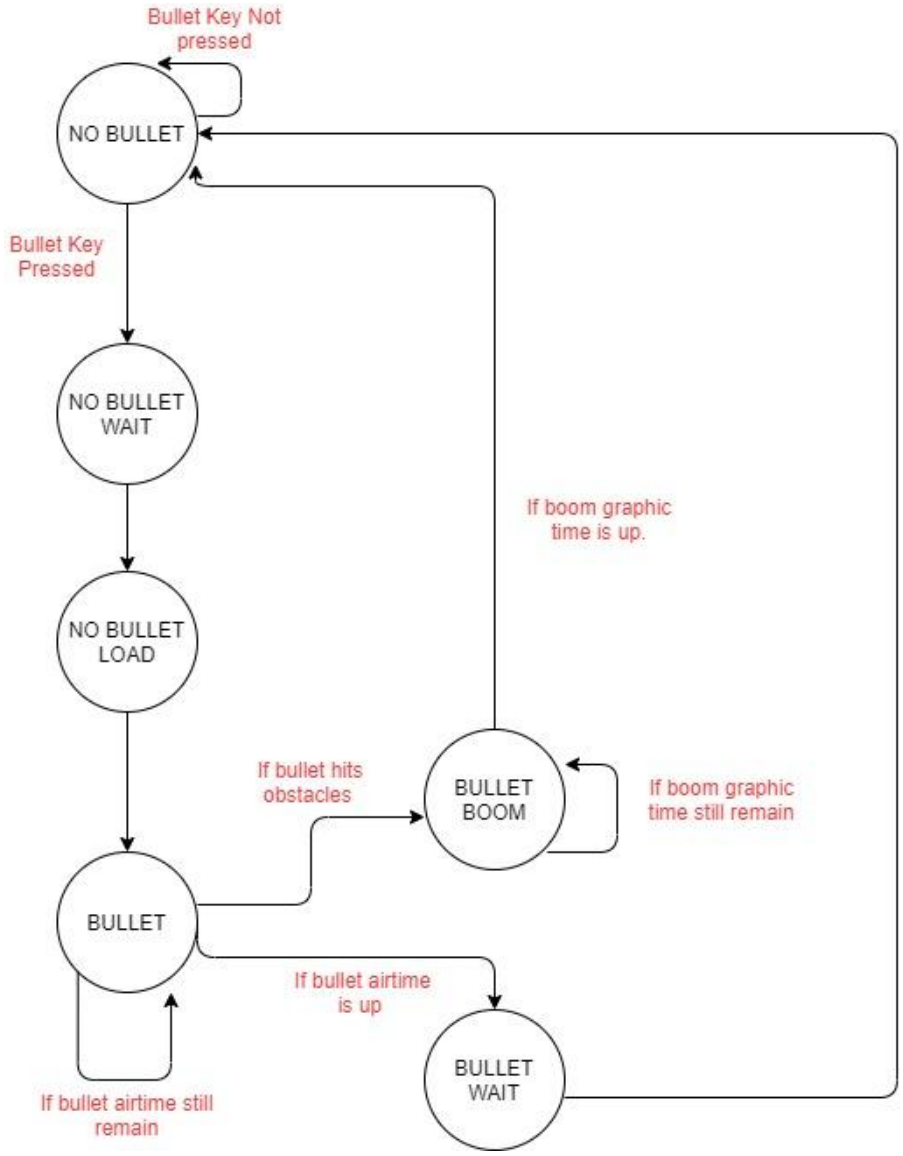
State Diagram of Select Modules

Top Level State Diagram



State diagram for our overall game. Details in each game state should have already been explained in the previous sections.

Bullet module state diagram



Explanation for each state

- **NO BULLET:** The start state for the bullet module. (i.e. if there is no bullet to be shot.)
- **NO BULLET WAIT / BULLET WAIT/ NO BULLET LOAD/ BOOM WAIT**: Unused states.
- **BULLET**: The state where we draw the bullet onto the display.
- **BULLET BOOM**: The state where we draw the boom graphics instead of the bullet for a short period of time.

Problems & Challenges Encountered

Final project is a huge challenge for both of us and we have encountered many problems while doing our final project from the beginning to the end.

Before doing the final project, COVID-19 prevents students from having face-to-face classes and going to university facilities. Limited resources and helps could be obtained by students. In this situation, we still need to have a high-quality final project that would already be a challenge for us. We only have the fpga board with no access to VGA monitor, keyboard suitable for the fpga, and other related equipment. Patterners no longer could stay together at the lab or study room to do the project together. My partner even bought a new computer and a new VGA monitor just for the final project. We also brought a VGA cable and a compatible keyboard. However, COVID-19 also limits the delivery speed, which costs us some time waiting for the equipment and cannot have them within a few days, which is really hard for some students staying on campus. However, I think we deal with this challenge pretty well and we have made a successful final project.

After beginning the final project, at first we do not only not only draw the picture onto the VGA screen using the sprite and palette, but also do not know how to move the position of different drawings, like tanks or bullets. However, we indeed did some research on it and figured out how to do similar things on it. Then we began to do the final project at a high speed rate.

In the middle state of doing the final project, our project has an option, which is two-players-mode. Thus, at least two keys should be able to be detected by the system at the same time. It is a problem for us because the lab 8 software part could only work for detection of only one key and no other team has an idea of how to deal with this situation. We communicated with many course assistants and we began to understand how to have multiple keys working at the same time. However, we still experimented many times since sometimes multiple keys are not working and four keys would be sorted automatically based on their ASCII values. Some keycode ASCII values are also too large and overflow would happen. As a result, we chose keys “w”, “a”, “s”, and “d” to move tank 1, “8”, “u”, “i”, and “o” to move tank 2, “c” to let tank 1 shoot , and “h” to let tank 2 shoot.

Last but not least, there are many trivial problems while writing systemverilog and we have no idea what to do with those problems but rewrite and try a new way of writing those codes. We have encountered these situations many times. Luckly, we finished the final project successfully.

One more thing to point out, though we failed to implement a successful audio in our final project, we tried to make it successful. However, we only have some sort of beats after we plug

in the headphones. Though the audio is not successful, we believe that we have a basic understanding of how audio drivers, audio controllers, and audio interfaces work on the FPGA. We believe that the reason why audio is not working is because we have not converted the wav music file into the correct hex file to represent the sound. After loading the file to FPGA, since the format is not correct, then it would not display the correct music but some sort of beats. One thing to point out is that our audio files do not affect our functionality at all but simply a new feature we want to add to the project.

Total Resources

LUT	12,210
DSP	40
MEMORY (BRAM)	3,207,169 bits
FLIP-FLOP	2813
FREQUENCY	128.9 Mhz
STATIC POWER	105.83mW
DYNAMIC POWER	0.75 mW
TOTAL POWER	177.95 mW

Conclusion

During the period of the final project, both of us collaborated perfectly and finished the final project up to our expectation. What is more, we also have learned many different useful techniques that could be implemented on FPGA. Double buffer and sprite sheet usage ensured that we could have a smooth and stable visual display on the VGA monitor. We also enhanced our ability not only writing system verilog for hardware, but also learned how to coordinate software and hardware together.

However, due to limitations of COVID-19, we have tried our best to improve our game and fix any possible bugs we would have. We also add as many unique features as possible. Though there are still many possible improvements that could be made in our final project, I think we have done a great job.

The final project has proved that we have a good and comprehensive ability of using FPGA and use it to accomplish certain tasks, which could be further utilized in the future. We could imagine that we could extend the knowledge of FPGA to some related fields and potentially have huge success.

Appendix

Customized Python Image converter script

Description: This script uses the python image library (PIL) and it supports image resizing, unique color counting, and conversion between image and text. This script was run on Jupyter Notebook.

```
from PIL import Image
# Opens the source image.
im = Image.open('image.png', 'r')
im.convert('RGB')
width, height = im.size
color = []
# load the pixel values of the original image.
pix_val = im.load()
new_pix_dict = {}
# create a new image with resized width and length.
im_new = Image.new('RGB', (new_width, new_height), 'black')
new_pix_val = im_new.load()

# Image resizing/scaling.
for y in range(new_height):
    for x in range(new_width):
        # clears the last four bits of each of the RGB channels by anding with 240.
        new_pix_val[x, y] = (pix_val[int(x/new_width*width), int(y/new_width*height)][0] & 240,
                             pix_val[int(x/new_width*width), int(y/new_height*height)][1] & 240,
                             pix_val[int(x/new_width*width), int(y/new_height*height)][2] & 240)
# save the converted image.
im_new.save('image_converted.png')

# translate the new pixel values into strings, and counting the number of unique colors.
for y in range(new_height):
    for x in range(new_width):
        new_pix_dict[x,y] = (format((int(new_pix_val[x, y][0] & 240) >> 4), 'x') +
                             format((int(new_pix_val[x, y][1] & 240) >> 4), 'x') +
                             format((int(new_pix_val[x, y][2] & 240) >> 4), 'x'))
        if new_pix_dict[x, y] not in color:
            color.append(new_pix_dict[x, y])

file = open('image.txt', 'w')
# write the strings, line by line, into a text file.
for y in range(new_height):
    for x in range(new_width):
        file.write(new_pix_dict[x, y]+'\n')
file.close()
```